Advances in Operating Systems CS60038
Term Project
Group - 13 Networking support in OS

Roll Number	**Name**	**Research Paper**
20CS10085	Pranav Mehrotra	**Eiffel**: efficient and flexible software packet scheduling, NSDI 2019
20CS30065	Saransh Sharma	**Shenango**: achieving high CPU efficiency for latency-sensitive datacenter workloads, NSDI 2019
20CS10088	Shah Dhruv Rajendrabhai	**TCP ≈ RDMA**: CPU-efficient remote storage access with i10, NSDI 2020
20CS10077	Vivek Jaiswal	**Loom**: flexible and efficient NIC packet scheduling, NSDI 2019

Topic: Eiffel: efficient and flexible software packet scheduling, NSDI 2019

a) **Problem Statement:**
The efficient and flexible scheduling of network packets is a critical concern in modern networks, impacting throughput, latency, and fairness. While software packet schedulers offer flexibility, they often suffer from inefficiency and inflexibility when compared to hardware solutions. These limitations have hindered the ability to adapt to different network requirements and emerging hardware platforms.

b) **Issues with Existing Solutions:**

1) <u>Inefficiency:</u> Software packet schedulers tend to be less efficient than their hardware counterparts due to the complexity of implementing scheduling algorithms in software. This inefficiency can introduce overhead and reduce overall performance.
2) <u>Inflexibility:</u> Many existing software packet schedulers are tightly coupled to specific hardware platforms, making it challenging to adapt them to new platforms or implement novel scheduling algorithms.
3) <u>Lack of Per-Flow Ranking:</u> Per-flow ranking and scheduling, which can enhance fairness and performance, are not well-supported in many existing software packet schedulers.

c) **Eiffel's Proposal:** The Eiffel system introduces a novel approach to address these issues, offering two core contributions:

1) <u>Efficiency:</u> Eiffel improves efficiency using integer priority queues and optimized scheduling algorithms. It utilizes integer priority queues, which operate in constant time (O(1)), as opposed to traditional priority queues (O(log n)). These queues are implemented using ranged buckets and simple bit manipulation operations. Eiffel also introduces Circular FFS-based Queue (cFFS) and Approximate Priority Queue (Gradient Queue) to facilitate O(1) enqueue/dequeue operations.
2) <u>Flexibility:</u> Eiffel enhances flexibility through innovative programming abstractions, allowing developers to define their scheduling algorithms and specify packet prioritization. This feature makes Eiffel adaptable to a wide range of scheduling policies.

d) **Design Features:**

1) <u>Integer Priority Queues</u>: Efficient integer priority queues are employed for packet scheduling.
2) <u>Optimized Scheduling Algorithms:</u> Eiffel offers a variety of scheduling algorithms tailored to specific network requirements, including throughput maximization, latency minimization, and fairness between flows.
3) <u>Novel Programming Abstractions:</u> Eiffel introduces novel programming abstractions that empower developers to create custom scheduling policies.

e) **Advantages:**

1) <u>Efficiency:</u> Eiffel's use of integer priority queues and optimized scheduling algorithms significantly improves efficiency in software packet scheduling.
2) <u>Flexibility:</u> Eiffel's novel programming abstractions provide greater flexibility, enabling developers to define and implement their scheduling policies.
3) <u>Per-Flow Ranking Support:</u> Eiffel supports per-flow ranking and scheduling, which enhances fairness and performance in packet scheduling.
4) <u>Adaptability:</u> Eiffel is more adaptable to different hardware platforms, reducing the constraints imposed by hardware-specific implementations.

f) **Limitations:**

The efficiency of Eiffel in the BESS network packet processing framework depends on two main parameters: batch size and queue size. Batching is important, but the impact of batching per flow varies with packet size. Small packets benefit from batching per flow to reduce overhead, while large packets perform well when batched together. Batching should be based on the expected traffic pattern, using Buffer modules when needed, this requires different configurations for each deployment and some pre-computations about the traffic patterns.

Title: Shenango - Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads

a) **Problem Statement:** Data center applications require microsecond-scale tail latencies and high CPU efficiency when responding to user requests involving multiple software services.

b) **Issues with existing solutions:**

   1) Low Latency vs. CPU Efficiency: Current operating systems compromise CPU efficiency for low latencies, while kernel-bypass network stacks waste CPU cycles.
   2) Core Allocation Overhead: Existing systems cannot efficiently reallocate cores on microsecond-scale adjustments. Current schedulers make load-balancing decisions at coarse granularities, limiting quick reactions.
   3) Difficulty in Load Estimation: Predicting instantaneous core requirements is challenging due to burst workloads and short timescales.
   4) Coarse-Grained Core Allocation: Inefficient Core Utilization: Existing systems leave a significant number of cores idle at all times to maintain low latency.

c) **Proposal Idea:** Shenango addresses the critical challenge of balancing high CPU efficiency with microsecond-scale tail latencies in data center applications by proposing a novel approach that includes core allocation strategies and efficient network handling.

d) **Design Features:** Shenango's design incorporates several innovative features to achieve its goals:

   1) Congestion Detection Algorithm: Shenango employs a congestion detection algorithm that considers both thread and packet queuing delays to make core allocation decisions. This fine-grained approach allows for swift responses to fluctuations in workloads.
   2) IOKernel: A central element of Shenango, the IOKernel, operates with root privileges and manages core allocation and network I/O. It performs core reallocations efficiently, reducing the associated overhead and ensuring rapid adjustments while maintaining low latencies.
   3) Runtime Optimization for Programmability: Shenango's runtime optimizes programmability by providing high-level abstractions like blocking TCP network sockets and lightweight threads. This design enhances developer flexibility and ease of use.
   4) Efficient Scheduling: The runtime's scheduling mechanism is designed for efficiency within applications across dynamically allocated cores. It uses per-kthread (kthread: kernel-level thread) run queues and work-stealing techniques to minimize tail latency, especially for workloads with variable service times.
   5) Network Handling: Shenango's runtime efficiently handles network functionality, including UDP and TCP protocol handling. It also provides mechanisms for packet reordering when required, ensuring the efficient delivery of network packets.

e) **Advantages:**

   1) Improved CPU Efficiency: Shenango balances low latency and high CPU efficiency, enabling data center applications to better use available resources.
   2) Rapid Core Allocation: The IOKernel efficiently manages core allocation, responding swiftly to fluctuations in workloads and avoiding compute congestion, resulting in minimal overhead.
   3) Programmability and Flexibility: The runtime's design prioritizes developer-friendliness, providing high-level abstractions and efficient programming interfaces for applications.
   4) Scalability: The design allows Shenango to scale to thousands of uthreads (uthread: User-level thread), making it suitable for data center applications with varying workloads.

f) **Limitations:**

   1) Workload Diversity: performance may vary significantly when dealing with extremely diverse workloads, making it challenging to manage applications with vastly different requirements efficiently.
   2) Hardware Dependency: The system's performance is influenced by hardware capabilities, such as the NIC's capacity to expose queuing buildup information to the IOKernel.

Title: TCP ≈ RDMA: CPU-efficient remote storage access with i10

a) **Problem Statement:** Addressing the Challenge of Remote Storage Access Efficiency, the paper delves into the challenge of achieving efficient remote storage access, given the ever-increasing speeds of internet and storage hardware. It highlights the critical issue of unsustainable CPU overheads in traditional remote storage access stacks.

b) **Issues with existing solutions:**

   1) Existing solutions for remote storage access suffer from significant CPU overhead, which adversely affects overall system performance.
   2) Many of these solutions require substantial modifications to applications and network stacks, making their adoption complex. Although new standards like NVMe-oF have emerged, efficiency concerns persist.

c) **i10 Proposals:** A Novel Kernel-Integrated Remote Storage Stack

   1) The paper introduces "i10", a pioneering remote storage stack implemented entirely within the Linux kernel.
   2) i10 serves as an intermediary layer that optimizes the exchange of control and data plane messages between the kernel storage stack and the network stack.

d) **Advantages of i10:**

   1) Seamless Integration: seamlessly integrates with existing applications without the need for modifications, simplifying its adoption by the industry.
   2) No Network Stack Changes: operates on the standard TCP/IP network stack, eliminating the requirement for alterations to network stacks or hardware.
   3) Compatibility with Modern Devices: adheres to the NVMe-oF standard, ensuring compatibility with emerging NVMe storage devices.
   4) Efficient Performance: can fully harness a 100Gbps link using standard server hardware while maintaining CPU efficiency similar to modern user-space and NVMe-over-RDMA solutions.

e) **Design Features:**

   1) i10 Lanes: Each (core, target) pair has a dedicated i10-lane, making use of per-core request queues in the block layer for efficient request management.
   2) i10 Caravan: It consolidates multiple requests into "caravans," reducing the need for sorting requests at the block layer and minimizing CPU overhead.
   3) Delayed Doorbells: It introduces the concept of "delayed doorbells" to mitigate context-switching overhead, especially in high-latency network conditions. This approach significantly improves performance under heavy loads.

f) **Implementation:**

   1) i10 is implemented in the Linux kernel, ensuring compatibility with commodity hardware and unmodified applications. The implementation optimizes data transmission by using kernel_sendmsg() for caravans, which leads to better CPU usage and throughput.
   2) Additionally, i10 offers a "no-delay path" for latency-sensitive requests. Parameters such as aggregation size and doorbell timeout are set at 16 and 50µs, respectively, to balance throughput and latency.
   3) To prevent caravan fragmentation, i10 uses fixed 8MB TCP buffers explicitly configured at session establishment.

g) **Testing and Limitations:**

   1) Testing:
      a) i10 demonstrates its ability to efficiently utilize a 100Gbps link with standard server hardware.
      b) It compares favorably with modern user-space and NVMe-over-RDMA solutions in terms of CPU efficiency.
   2) Limitations: The paper does not extensively address the impact of varying network conditions on i10's performance.

Title: Enhancing Network Interface Card (NIC) Performance with Loom

a) **Problem Statement:**

Modern data centers face challenges in efficiently enforcing high-level hierarchical network policies due to the limitations of current multi-queue Network Interface Cards (NICs). Existing NICs allocate the responsibility of packet scheduling decisions to the NIC itself, leading to a lack of flexibility and control from the operating systems (OSes). Additionally, these NICs rely on static and inflexible on-NIC packet schedulers, making it difficult to ensure effective traffic isolation among competing applications with diverse network objectives.

b) **Issues with existing solutions:** Existing NICs suffer from two primary limitations:

1) <u>NIC-Centric Decision Making:</u> Current NICs make all packet scheduling decisions, rendering OSes incapable of precise control over network prioritization due to the requirement of multiple independent NIC transmit queues for achieving line-rate performance.
2) <u>Static Packet Schedulers:</u> On-NIC packet schedulers are rigid, supporting only a limited number of traffic classes and scheduling algorithms. This deficiency results in the inability to guarantee optimal performance for latency-sensitive and bandwidth-intensive applications simultaneously.

c) **Loom's Proposal:** To overcome these limitations, this paper introduces a novel NIC design, Loom, which redistributes the responsibility for per-flow scheduling decisions from the OS to the NIC. Loom's approach achieves greater efficiency by reducing OS overhead and improving resource allocation.

1) <u>Restricted Directed Acyclic Graph (DAG) Policy:</u>
    a) Loom introduces a policy abstraction based on a Restricted Directed Acyclic Graph.
    b) The design enforces a tree-like structure when shaping nodes are removed, ensuring that packets are not reordered, and per-destination and per-path rate limits can be maintained effectively.

2) <u>Programmable Hierarchical Packet Scheduler:</u>
    a) Loom leverages switch design principles to introduce a programmable hierarchical packet scheduler, allowing for the customization of scheduling algorithms.
    b) The scheduler operates through the enqueue and dequeue functions and maintains a local state to determine packet priority.
    c) The versatility of design permits the implementation of various scheduling algorithms such as Strict Priority Scheduler, Weighted Fair Queuing (WFQ), and Least Slack Time First Scheduler, based on the application's needs.

d) **The uniqueness of Loom:** Loom's uniqueness lies in its ability to centralize packet scheduling within the NIC, granting the OS precise control over prioritization while maintaining high line rates. By adhering to a Restricted DAG Policy and introducing a programmable scheduler, Loom achieves efficient and flexible network management, addressing the limitations of existing NICs.

e) **Limitations:**

i) <u>Complexity:</u> Implementing a programmable hierarchical packet scheduler and adopting the Restricted DAG Policy might introduce additional complexity, which could impact the ease of deployment and maintenance.

ii) <u>Compatibility:</u> Compatibility with existing network infrastructure and applications may require further consideration and adaptation.

iii) <u>Resource Overhead:</u> The effectiveness of Loom in reducing OS overhead may vary depending on specific network configurations and workloads.

f) **Conclusion**: Loom's design offers a promising solution to the challenges of efficient network policy enforcement in data centers, allowing for more flexible and precise control of packet scheduling while maintaining high-performance standards.